LIBRARY
RESEARCH REPORTS DIVISION
NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA 93940

NPS52-84-013

NAVAL POSTGRADUATE SCHOOL

Monterey, California

REAL-TIME CONTOUR SURFACE DISPLAY GENERATION

Michael J. Zyda

September 1984

Prepared for:

Chief of Naval Research
Arlington, VA 22217

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Commodore R. H. Shumaker
Superintendent

D. A. Schrady
Provost

The work reported herein was supported in part by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research.

This report was prepared by:

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NPS52-84-013	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) REAL-TIME CONTOUR SURFACE DISPLAY GENERATION		5. TYPE OF REPORT & PERIOD COVERED 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Michael J. Zyda		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93943		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61152N; RR000-01-10 N0001484WR41001
11. CONTROLLING OFFICE NAME AND ADDRESS Chief of Naval Research Arlington, VA 22217		12. REPORT DATE September 1984 13. NUMBER OF PAGES 21
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Algorithms, architecture, contouring, contouring tree, contour surface display generation, real-time display generation

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

We present in this study the architectural specification and feasibility determination for a real-time contour surface display generator. We begin by examining a recently reported, highly decomposable algorithm for contour surface display generation. We establish a piece of the total algorithm as the algorithm component. The algorithm component is that part of the algorithm that can be executed in parallel, independently from the computations performed on any other algorithm subpart. We propose an architecture for the algorithm component, and model that architecture in order to determine the real-time capability of the

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

algorithm. We then model the larger system of multiple algorithm component processors. This modeling effort is performed with respect to a particular application requiring real-time contour surface display generation. A VLSI feasibility computation is then performed on the proposed architecture. The study ends with a look at the impact of real-time contour surface display generation on the design of the graphics display system.

Real-Time Contour Surface Display Generation ‡

Michael J. Zyda

Naval Postgraduate School,
Code 52, Dept. of Computer Science,
Monterey, California 93943

ABSTRACT

We present in this study the architectural specification and feasibility determination for a real-time contour surface display generator. We begin by examining a recently reported, highly decomposable algorithm for contour surface display generation. We establish a piece of the total algorithm as the algorithm component. The algorithm component is that part of the algorithm that can be executed in parallel, independently from the computations performed on any other algorithm subpart. We propose an architecture for the algorithm component, and model that architecture in order to determine the real-time capability of the algorithm. We then model the larger system of multiple algorithm component processors. This modeling effort is performed with respect to a particular application requiring real-time contour surface display generation. A VLSI feasibility computation is then performed on the proposed architecture. The study ends with a look at the impact of real-time contour surface display generation on the design of the graphics display system.

Categories and Subject Descriptors: I.3.1 [**Hardware Architecture**]: architectures, parallel processing, VLSI implementations; I.3.2 [**Graphics Systems**]: multiprocessing systems; I.3.3 [**Picture/Image Generation**]: surface visualization; I.3.5 [**Computational Geometry and Object Modeling**]: data structures, discrete planar contours, modeling molecules, surface approximation, surface generation, surface representation, surfaces, 3D graphics; I.3.6 [**Methodology and Techniques**]: contouring, interactive systems, parallel processing; I.3.7 [**Three-Dimensional Graphics and Realism**]: line drawings, line generation algorithms, real-time graphics, surface plotting, surface visualization, surfaces; I.3.m [**Miscellaneous**]: VLSI;

General Terms: Algorithms, architecture;

Additional Key Words and Phrases: contouring, contouring tree, contour surface display generation, real-time display generation;

‡ This work has been supported by the NPS Foundation Research Program.

1. Introduction

Contour surface display generation is one of the most frequently used graphics algorithms [Barry,1979], [Faber,1979], [Wright,1979], [Zyda,1984a], [Zyda,1984b], [Zyda,1983], [Zyda,1982], [Zyda,1981]. A contour surface display is a visual representation of a surface by the collection of lines formed when that surface is intersected by a set of parallel planes. The lines formed on each of those planes are called contours. A contour represents the set of points that belong to both the surface and the particular intersecting plane. Contour surface displays are used in X-ray crystallography, computer-aided tomography, and other applications for which grid data is collected. Contour surface display generation is generally depicted as a computationally slow operation whose output is sent to a plotter or film recorder. A number of papers have been written documenting "breakthroughs" that increase the speed of contour surface display generation. One author has reported that his contour surface display generation subroutine used one second of central processor time on NCAR's Control Data 7600 [Wright,1979]. Although a contour surface display generation program of this speed is useful for static situations, it is found to be lacking for interactive applications that generate a succession of contour surface displays in response to contour level changes read from a control dial.

Interactive applications that cause the generation of a succession of images require that the human intervention be acknowledged by a visual change to the current display within a finite element of time, called real-time. For a system that generates a new contour surface display in response to human intervention, real-time means that we must be able to produce and distribute a new picture in the amount of time it takes the graphics hardware to change display frames. This is typically one-thirtieth of a second. Any greater amount of time is discernable by the viewer, either as a flicker or a hesitation in the picture update. In fact, one-thirtieth of a second is discernable to many people, making one-sixtieth of a second a more desirable time for the change of display frames [Newman,1979].

One application in which real-time contour surface display generation is important is the determination of molecular structures from the electron density data generated by X-ray crystallography [Barry,1979]. Such an operation is executed interactively by using a computer graphics program that displays a Dreiding (stick) model of the molecule, inside a contour surface display of the corresponding region of the molecule's electron density grid. In addition to the graphics function, the computer program monitors a series of signals generated by the user, while the user is turning the various knobs on a control console [Zyda,1980]. The values read from these knobs are interpreted by the program as modifications to either the molecule or the surface display. Modifications to the molecule take the form of bond rotations or bond lengthenings. Modifications to the contour surface display take the form of an increase or decrease of the contour level. The goal of this process is to produce the stick model of the molecule that best fits inside the given electron density data set. The user can determine whether or not the model fits the density grid by modifying the contour level, shrinking the contour surface to the molecule. Similarly, the user can expand the contour surface from the stick model for better visibility. This function requires that the hardware have the capability to rapidly change the contour display as its contour level changes.

We know from [Zyda,1984a] that the generation of a contour surface display, such as those required by the above application, cannot be accomplished in real-time using a conventional uniprocessor. This failure is due to the fact that contour surface display generation algorithms require many more

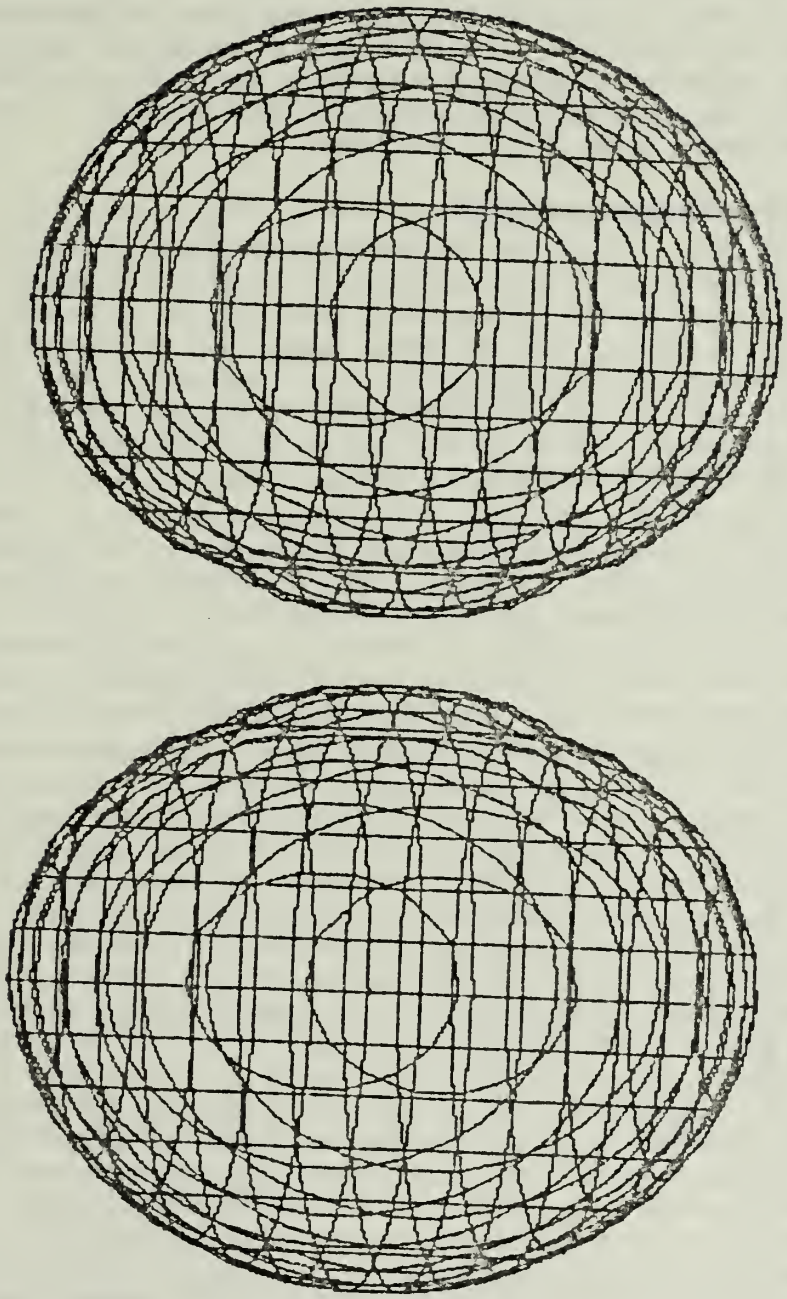


Figure 1
Contour Surface Display Generated from a Hydrogen Atom
Wavefunction Squared (3dxy orbital)

instructions executed per second than can be provided by currently available uniprocessors. In the past, this limitation of the conventional processor has relegated such applications to either the non real-time environment (waiting a few minutes for each display), or to the equally unsatisfying environment of motion picture film. Because of this, this study looks for multiprocessor solutions to the real-time contour surface display generation problem. At the present time, efficient multiprocessor solutions generally mean VLSI solutions. Consequently, the multiprocessor architectures examined in this study are those implementable in the VLSI technologies.

2. Definitions and Decomposability

A contour surface is a visual display that represents all points in a particular region of three-space <x,y,z> which satisfy the relation f(<x,y,z>)=k, where k is a constant known as the contour level. The function f represents a physical quantity which is defined over the three-dimensional volume of interest. The visual display created by this algorithm is the collection of lines that belong to the intersection of both the set of points that satisfy the relation f(<x,y,z>)=k, and a set of regularly spaced parallel planes that pass through the region of three-space for which the relation is defined.

For this study, the function f is approximated by a discrete, three-dimensional grid created by sampling that function over the volume of interest. The three-dimensional grid contains a value at each of its defined points that corresponds to the physical quantity obtained from the function, i.e. the value associated with point (x_0,y_0,z_0) is v_0, where $f(x_0,y_0,z_0)=v_0$. In order to minimize confusion, we will specify the value at a particular grid point (x,y,z) by a(x,y,z), and will specify the value at a particular point (x,y,z) of the function by f(x,y,z).

The visual display of the contour surface is created from this three-dimensional grid by taking two-dimensional slices of the grid, and constructing the two-dimensional, planar contours for each slice at the designated contour level. A slice of a three-dimensional grid is a planar, orthogonal, two-dimensional grid assigned a constant coordinate in three-space, i.e. an x-y slice of a(<x,y,z>) corresponds notationally to a(<x,y>) for a particular z coordinate. The two-dimensional, planar contours created are the lines that satisfy the relation a(<x,y,z>)=k for a particular planar coordinate, either x, y, or z, where again k is the constant contour level. If we contour all x-y slices of the three-dimensional grid at contour level k, we will have a stack of parallel contours approximating the contour surface, each planar set of contours corresponding to a particular z coordinate. If we contour all x-z slices of the three dimensional grid, we again will have a stack of parallel contours approximating the contour surface, each planar set of contours corresponding to a particular y coordinate. Likewise, if we contour all y-z slices of the three-dimensional grid, we will have a stack of parallel contours approximating the contour surface, each planar set of contours corresponding to a particular x coordinate. The assemblage of the three sets of parallel, planar contours, i.e. the simultaneous display of all the contours created for the x-y, x-z, and y-z planes of the three-dimensional grid, produces a "chicken-wire-like" contour surface display (see Figure 1). The three-dimensional contour surface display described in this study is created by such a procedure.

A decomposable algorithm for contour surface display generation has been described in [Zyda,1984b]. That algorithm is constructed from a two-dimensional contouring algorithm that is used to contour all the possible planar, orthogonal, two-dimensional grids of a larger three-dimensional grid. The two-dimensional contouring algorithm of that paper is comprised of components,

Figure 2
Example Contour Grid with Contours Drawn for Level 50

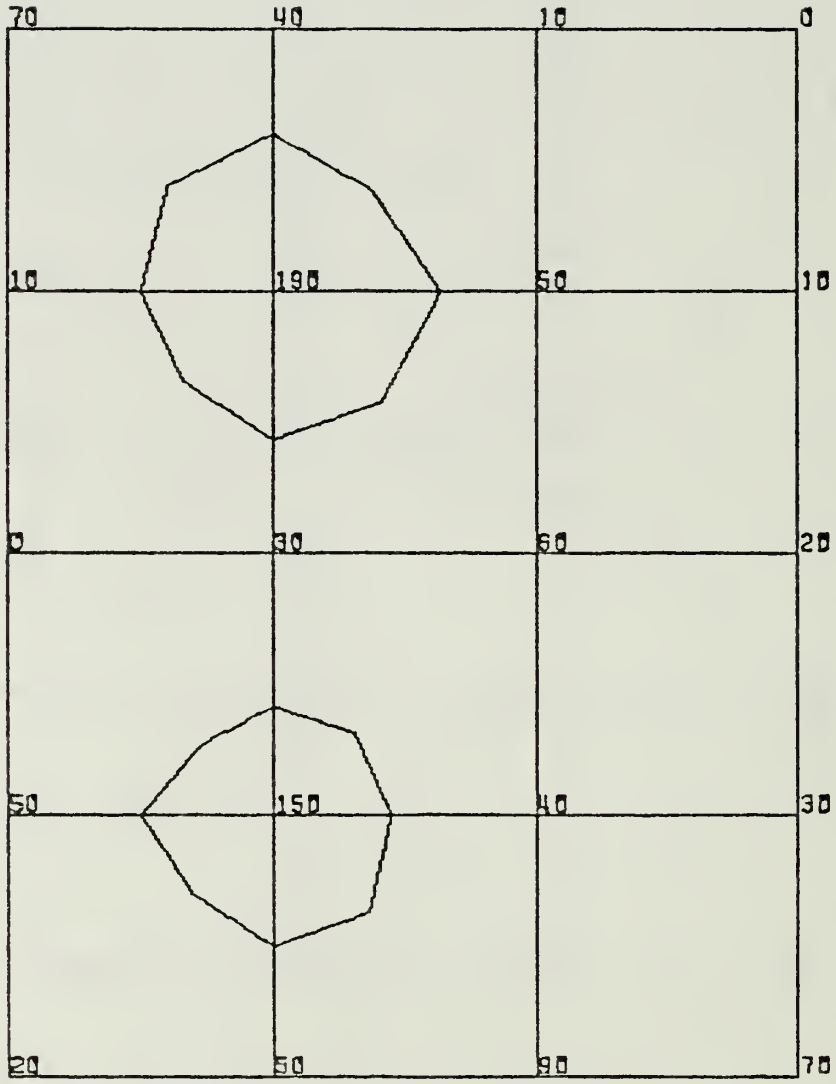


Figure 3
Example Contour Grid with Contours Drawn for Level 100

called algorithm components, that operate on individual 2 x 2 subgrids of a larger two-dimensional grid. In the algorithm, the computations necessary for generating the contour lines for a single 2 x 2 subgrid are independent from those required for any other 2 x 2 subgrid. (Note: a 2 x 2 subgrid is defined to be that portion of the two-dimensional grid bounded by four adjacent grid points. In the two-dimensional grid of Figure 2, the lower, lefthand 2 x 2 subgrid is bounded by points (1,1), (2,1), (2,2), and (1,2).) If we compute the contours corresponding to contour level k for all 2 x 2 subgrids of a two-dimensional grid, then we will have determined the complete set of contours for that grid. If we compute the contours corresponding to contour level k for all possible 2 x 2 subgrids of the larger three-dimensional grid, then we will have the complete contour surface display for that grid. We use this formulation for the contouring algorithm in this study.

3. The Contouring Tree

The contouring algorithm in [Zyda,1984b] is based upon a data structure called the contouring tree. A contouring tree represents the edge value relationships of a 2 x 2 subgrid in a form that permits the rapid generation of the contour display for any contour level contained within the represented subgrid (see Figure 4). The formulation of the contouring tree is based upon the observation that for any two-dimensional grid a continuous series of contour displays can be created for contour levels in the range of the minimum and maximum grid values (see Figure 5, and [Zyda,1984a], [Zyda,1984b], [Zyda,1983], [Zyda,1982], [Zyda,1981]).

The use of the contouring tree is outlined best with an example of a small two-dimensional grid. Figures 2 and 3 depict the contours generated for contour levels 50 and 100. The contours at level 100 are closed contours, forming simple, connected loops. The contours at level 50 are open contours. Figures 4 and 6 present the contouring trees created for two 2 x 2 subgrids of the 4 x 5 plane. The edges of the contouring trees correspond to the directed, downhill edges inscribed on the 2 x 2 subgrids of the figures. There are eight directed edges on each subgrid, four for the boundary edges and four for the edges to the subgrid's center point. The value used for the center point is the average of the four values comprising the corners of the 2 x 2 subgrid. (A reference as to the usefulness of the center point average value in generating smooth contours is found in [Sutcliffe,1980].) The edges of the contouring trees are ordered, maintaining the same counterclockwise ordering as in the original subgrids. A "1" under a node indicates that a setpoint display command should be generated for any coordinate that is created along an edge that has that connectivity on its lower valued node. A "0" indicates a drawto display command in a similar fashion and a "2" indicates a drawpoint.

Display generation from a contouring tree is accomplished by performing a pre-order traversal of that contouring tree, producing a coordinate and drawing instruction whenever the desired contour level is found to be within the range of an edge of the contouring tree. A pre-order traversal visits the root, the left subtree, the middle subtree, and then the right subtree. An edge's range is defined to be the set of values between those associated with the nodes on either end of the edge. More precisely, we say a contour level is within an edge if the following condition holds:

lower_node's_value $\leq$ contour_level $<$ higher_node's_value

For example, in Figure 4a at contour level 100, we issue coordinates and drawing instructions for the edges (2,2)-(3,2), (2,2)-(2.5,2.5), and (2,2)-(2,3). The drawing

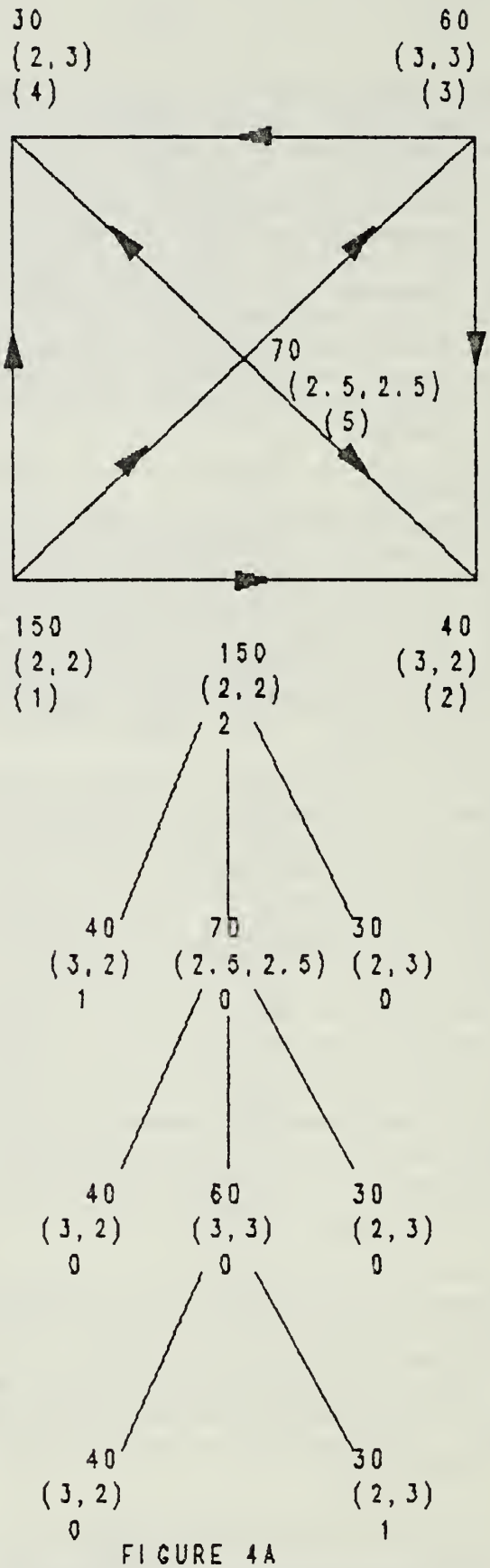


FIGURE 4A

SAMPLE CONTOURING TREE FOR A 2 X 2 SUBGRID

Level 50

X	Y	Z	D
2.9091	2.0000	1.0000	1
2.8333	2.1667	1.0000	0
3.0000	2.5000	1.0000	0
2.6667	3.0000	1.0000	1
2.2500	2.7500	1.0000	0
2.0000	2.8333	1.0000	0

Level 100

X	Y	Z	D
2.4545	2.0000	1.0000	1
2.3125	2.3125	1.0000	0
2.0000	2.4167	1.0000	0

Column D is the drawing command, ie. 1 = SETPOINT, 0 = DRAWTO.

Figure 4b
Coordinates Generated for Sample 2 x 2 Subgrid

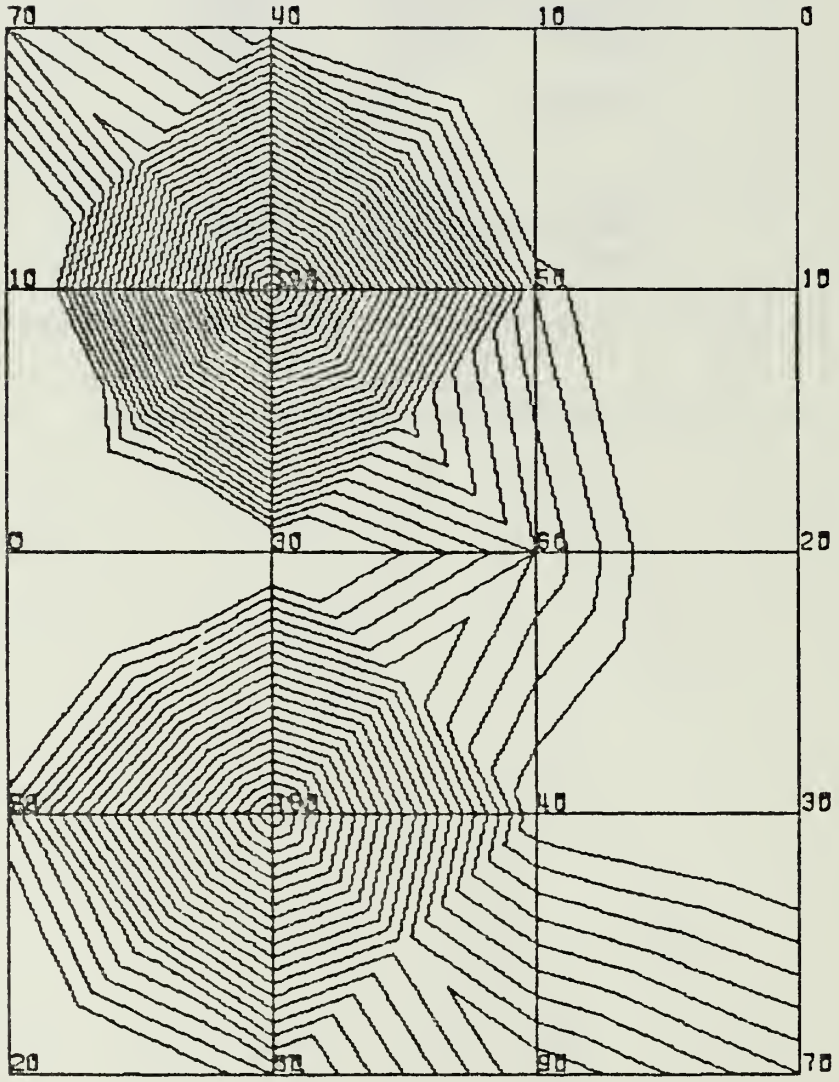


Figure 5
Example Contour Grid with Contours Drawn for Multiple Contour Levels

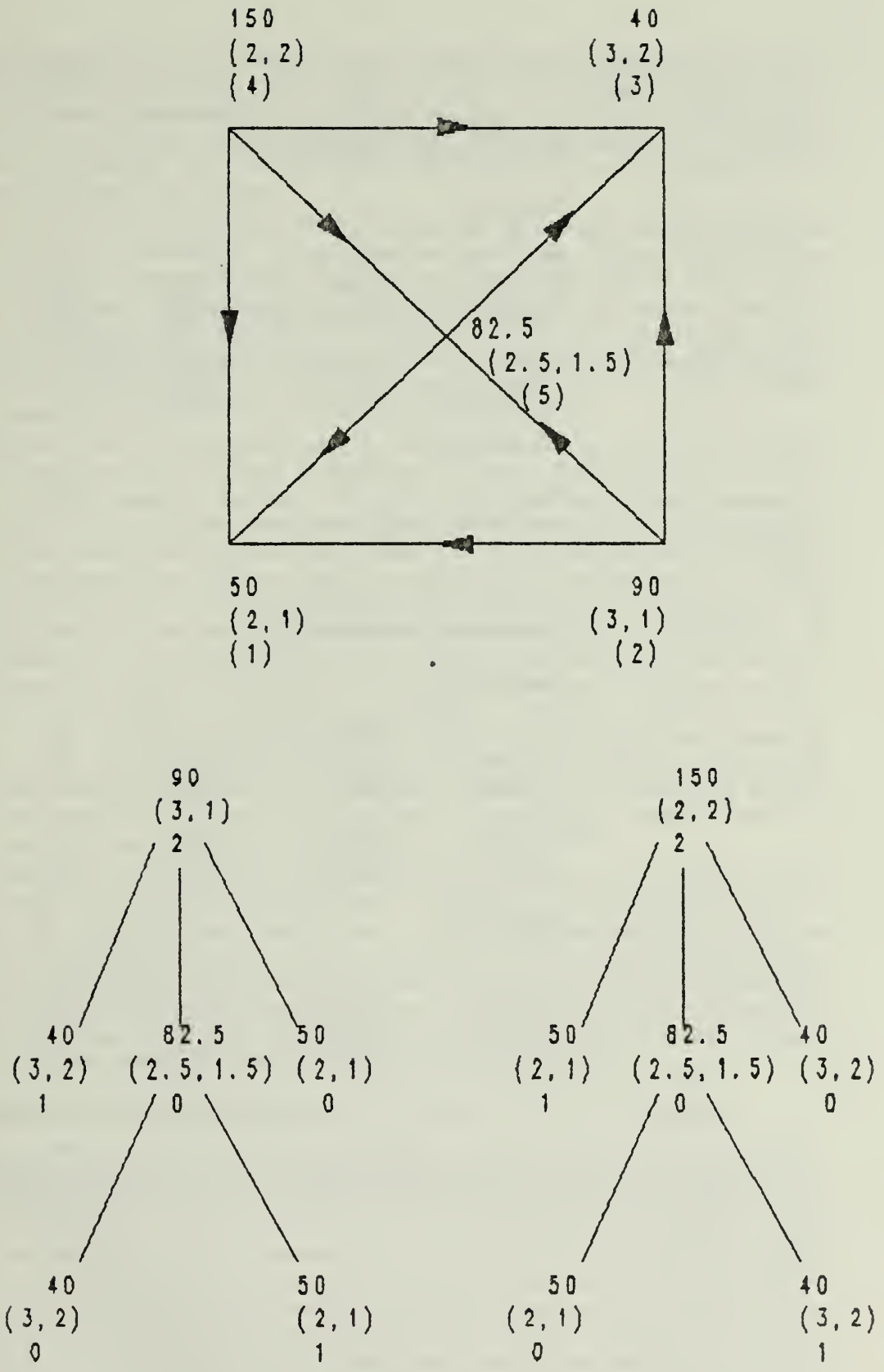


FIGURE 6A
SAMPLE CONTOURING TREE FOR A 2 X 2 SUBGRID WITH SADDLE POINT

Tree rooted at value 90

Level 50

X	Y	Z	D
3.0000	1.8000	1.0000	1
2.8824	1.8824	1.0000	0
2.0000	1.0000	1.0000	1
2.0000	1.0000	1.0000	0

Level 100

X	Y	Z	D
no coordinates generated			

Tree rooted at value 150

Level 50

X	Y	Z	D
2.0000	1.0000	1.0000	1
2.0000	1.0000	1.0000	0
2.8824	1.8824	1.0000	1
2.9091	2.0000	1.0000	0

Level 100

X	Y	Z	D
2.0000	1.5000	1.0000	1
2.3704	1.6296	1.0000	0
2.4545	2.0000	1.0000	0

Column D is the drawing command, ie. 1 = SETPOINT, 0 = DRAWTO.

Figure 6b
Coordinates Generated for Sample 2 x 2 Subgrid with Saddle Point

instruction issued for each of these edges is again the one associated with the lower valued node of the edge. The coordinate for each of these edges is generated by a linear interpolation of the edge's endpoint coordinates according to the decrease in contour level along the edge. The coordinates and drawing instructions generated for the contouring trees of Figures 4a and 6a are represented in Figures 4b and 6b.

There are some subtleties not evident from the above that are best detailed using a pseudocode description of the traversal algorithm. Figure 7 depicts the traversal procedure for the contouring tree assuming a particular data organization. The notation is quite standard. The pointers to the descendent nodes of NODE are LEFT(NODE), MIDDLE(NODE), and RIGHT(NODE). For each node of the contouring tree, there are three pieces of information: the value associated with the node, VALUE(NODE), the coordinate associated with the node, XYZ(NODE), and the connectivity associated with the node, CONN(NODE).

The generation of coordinates and drawing instructions from a contouring tree begins with routine CONTOUR_SUBGRID of Figure 7. That routine receives a pointer to the root node of the contouring tree. It then starts the traversal by calling routine VISIT with that root node. Routine VISIT checks to see if the edge defined by the passed in node and that node's ancestor, NODE and ANCESTOR, contains the contour level. If the edge does contain the contour level, the edge intersection coordinate is computed using linear interpolation and issued to the display along with the connectivity associated with that node, CONN(NODE). If we issue a coordinate and connectivity for a node, we need to check the subtree under that node for equivalued edges. If an equivalued edge at the contour level is found, a coordinate and drawing instruction pair are issued for that equivalued edge (routine VISIT_SUBTREE). Once a coordinate and drawing instruction pair have been issued for an edge, and once the subtree beneath that edge has been investigated for equivalued edges, further traversal of that subtree is terminated. If an edge is found not to contain the contour level, the traversal continues as depicted at the bottom of routine VISIT.

The pre-order traversal procedure described generates the coordinates and drawing instructions for the part of the 2 x 2 subgrid the contouring tree represents. To generate the coordinates for a larger two-dimensional grid, we generate the contouring trees for each 2 x 2 subgrid of that grid, and then apply the traversal procedure to those trees. We note here that no ordering is required in the generation of coordinates for the 2 x 2 subgrids. The coordinate and drawing instruction set generated for each 2 x 2 subgrid is complete and independent of the picture generated for any neighboring 2 x 2.

3.1. Contouring Tree Use Discussion

Having presented the use of the contouring tree, we must discuss its limitations. The initial impression is that the contouring tree provides a nice, uniform framework for generating the coordinates and drawing instructions appropriate to the 2 x 2 subgrid. This is close to correct but there are problems. These problems all concern issues of picture efficiency. Since the display generated for each 2 x 2 subgrid is generated independently of any neighboring 2 x 2 subgrids, equivalued lines at the contour level on the border of a subgrid will be duplicated. A similar problem occurs for subgrid corner values that equal the contour level. If we display either of the above cases on a calligraphic display device, we will see a bright line for the equivalued edge, and a bright point for the grid value equal to the contour level. Another problem, also due to the independent computation of each 2 x 2 subgrid, is that no ordering is provided for coordinates that come out of this algorithm. For calligraphic displays, this is

Contouring Tree Description

Pointers to descendent nodes:

```
LEFT(NODE)
MIDDLE(NODE)
RIGHT(NODE)
```

Values associated with each node:

```
VALUE(NODE):  grid value
XYZ(NODE)  :  coordinate of that grid value.
CONN(NODE) :  drawing instruction.
```

```
Procedure CONTOUR_SUBGRID(ROOT)

  VISIT(ROOT,ROOT)  # begin the traversal of the pointed at
                    # contouring tree.

end.
```

```
Procedure VISIT(NODE,ANCESTOR)

  if(NODE == NULL)
  {
    return
  }

  if((VALUE(NODE) <= CONTOUR_LEVEL < VALUE(ANCESTOR))
     OR
    (VALUE(NODE) == CONTOUR_LEVEL AND NODE == ANCESTOR))
  {

    # Edge contains the contour level.

    Issue a coordinate computed via linear interpolation
    along the edge.

    Issue CONN(NODE) as the drawing instruction.
```

Figure 7
Pseudocode of the Traversal Algorithm for the Contouring Tree

```
    # Check subtrees of this node for equivalued edges.
    VISIT_SUBTREE(LEFT(NODE),NODE)
    VISIT_SUBTREE(MIDDLE(NODE),NODE)
    VISIT_SUBTREE(RIGHT(NODE),NODE)

    return   # no need to examine the subtree further.

  }  # endif coordinates were generated for an edge.


  VISIT(LEFT(NODE),NODE)   # visit left subtree.
  VISIT(MIDDLE(NODE),NODE)   # visit middle subtree.
  VISIT(RIGHT(NODE),NODE)   # visit right subtree.

  return

end



Procedure VISIT_SUBTREE(SUBNODE,SUBANCESTOR)

  if(SUBNODE == NULL)
  {
    return
  }

  if(VALUE(SUBNODE) == CONTOUR_LEVEL)
  {

    Issue coordinates for the equivalued edge.
    Setpoint on XYZ(SUBANCESTOR).
    Drawto     XYZ(SUBNODE).

  }


  VISIT_SUBTREE(LEFT(SUBNODE),SUBNODE)
  VISIT_SUBTREE(MIDDLE(SUBNODE),SUBNODE)
  VISIT_SUBTREE(RIGHT(SUBNODE),SUBNODE)

  return

end
```

Figure 7 (continued)
Pseudocode of the Traversal Algorithm for the Contouring Tree

a problem because for such devices electron beam movement is expensive. A contour display that causes the maximum movement of the electron beam every other subgrid greatly decreases the the vector capability of the calligraphic display device.

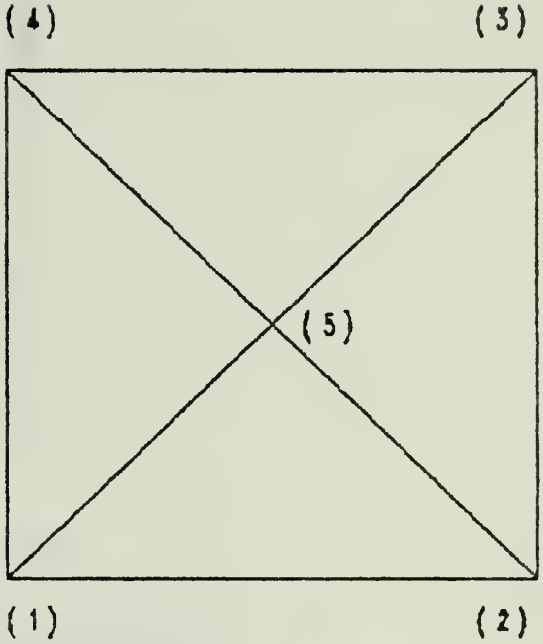
There are three possible solutions to the first problem, that of duplicate vectors. The easiest solution is to choose an output display device for which such picture inefficiencies do not matter, i.e. a raster display. Vector ordering is also eliminated as a problem with this solution. The second solution to the vector duplication problem is to set aside points and lines at the contour level that correspond to subgrid boundaries. A final pass at the end of the computation for a complete two-dimensional plane could readily cull the duplicates. This second solution does nothing for the vector ordering problem. This solution also requires a join operation on the results of the algorithm component computations for each two-dimensional grid, and consequently, diminishes the algorithm's concurrency potential. The third solution, and the most expensive of the three, is to merge the set of trees generated for the two-dimensional grid such that duplicate edges in separate trees are eliminated. This solution has the added benefit that the resultant contours are generated in an order that solves the beam movement problem. This solution is not described in detail here and the reader is referred to [Zyda,1981] for further detail. For this study, the first and simplest solution is assumed for purposes of maximizing the concurrency potential of the algorithm. Consequently, the expected output display device is the raster display.

3.2. Contouring Algorithm Simplifications

Before we look in detail at a special architecture for computing the contour lines for a 2 x 2 subgrid, we first consider simplifications to that algorithm that greatly increase its speed. The first simplification we consider is one that eliminates contouring tree construction for the 2 x 2 subgrid. In [Zyda,1984a], a procedure for contouring tree construction is described. That procedure begins with the composition of a 5 x 5 adjacency matrix that represents the directed graph of the edges inscribed on the four grid points and center average value point of the 2 x 2 subgrid. Using a 5 x 5 adjacency matrix to describe a graph that has a constant set of eight edges, whose only changes are in the directions of those edges, is quite expensive. We can replace that adjacency matrix by a field of eight bits, with a one indicating one direction and a zero the other. This replacement makes quite clear the fact that there are really only 256 possible configurations of contouring trees. If we remember that the center point is not ever chosen as maxima, and that subgrid digraphs without maxima have no contouring trees, this reduces the total to 120 possible configurations of contouring trees [Zyda,1984a]. With these simplifications, we can look up the tree configuration for a 2 x 2 subgrid from a small table once we have its configuration number. The configuration number is composed by an assignment of edges and directions to each bit of the eight bit number (see Figure 8).

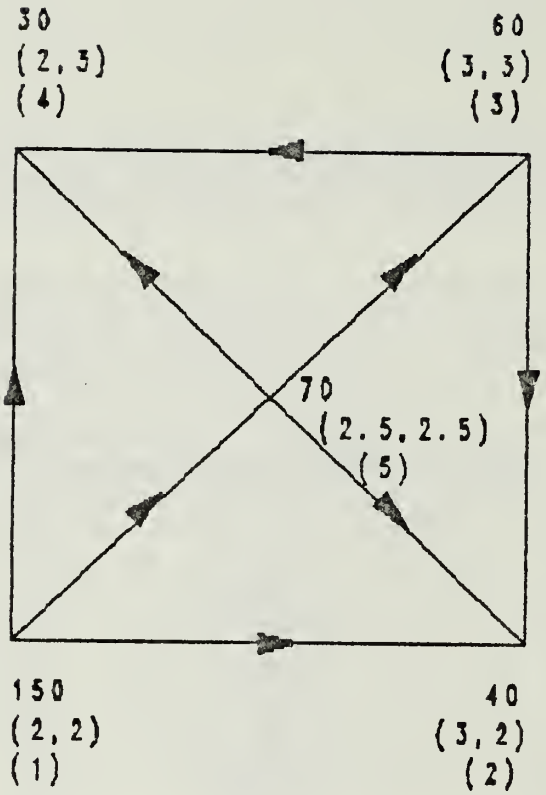
The second simplification we consider is one that speeds up the use of the contouring tree in its generation of the contours. The time consuming portion of this process is the traversal of the contouring tree. One speed up is to precompute the tree traversals for each contouring tree by forming a linear list of each tree's edges in traversal order. The data necessary for the contouring trees represented in this form for the example trees of Figures 4 and 6 can be seen in Figures 9 and 10. The traversal is accomplished by stepping through the linear list of edges using the same edge evaluation scheme as described previously, i.e. a contour level is within an edge (and hence a coordinate should be generated) if:

BIT NUMBER	EDGE REPRESENTED
0	1 → 2
1	2 → 3
2	3 → 4
3	4 → 1
4	1 → 5
5	2 → 5
6	3 → 5
7	4 → 5



A ONE IN THE BIT POSITION MEANS THE EDGE EXISTS.
A ZERO IN THE BIT POSITION MEANS THE EDGE OF OPPOSITE DIRECTION EXISTS.

FIGURE 8
CONFIGURATION NUMBER EDGE ASSIGNMENTS FOR THE 2 X 2 SUBGRID

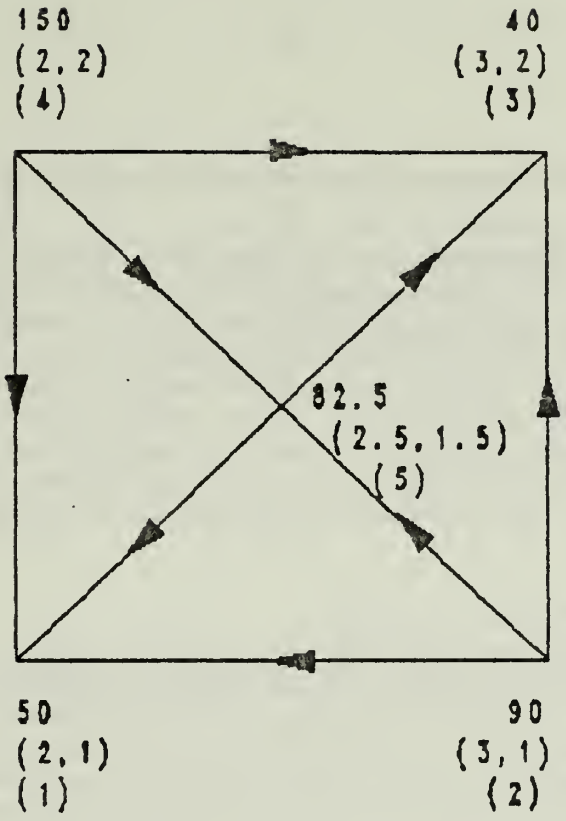


Configuration Number = 21 = 0001 0101

Tree Number 1 has 9 edges.

Edge #	Current Node's Subgrid#	Previous Node's Subgrid#	Next Edge if coord. is gen'd.	Connectivity
1	1	1	10	2
2	2	1	3	1
3	5	1	7	0
4	2	5	5	0
5	3	5	7	0
6	2	3	7	0
7	4	3	8	1
8	4	5	9	0
9	4	1	10	0

Figure 9
Traversal List Representation of the Contouring Tree of Figure 4



Configuration Number = 170 = 1010 1010
Tree Number 1 has 6 edges.

Edge #	Current Node's Subgrid#	Previous Node's Subgrid#	Next Edge if coord. is gen'd.	Connectivity
1	2	2	7	2
2	3	2	3	1
3	5	2	5	0
4	3	5	5	0
5	1	5	6	1
6	1	2	7	0

Configuration Number = 170 = 1010 1010
Tree Number 2 has 6 edges.

Edge #	Current Node's Subgrid#	Previous Node's Subgrid#	Next Edge if coord. is gen'd.	Connectivity
1	4	4	7	2
2	1	4	3	1
3	5	4	5	0
4	1	5	5	0
5	3	5	6	1
6	3	4	7	0

Figure 10
Traversal List Representation of the Contouring Tree of Figure 6

current_node's_value ≤ contour_level < previous_node's_value

If a coordinate is generated for an edge, the subtree delineated by the "next edge" field of the table is examined for equivalued edges at the contour level. If such equivalued edges are encountered, coordinates and drawing instructions appropriate to that edge are generated. Note that the traversal list tables of Figures 9 and 10 are in terms of the subgrid numbering scheme rather than in terms of explicit grid values. In the design of the architecture for the contour surface display generator, we use the configuration number to find the traversal list for the contouring tree, and use that traversal list to generate the display coordinates. This is instead of actually constructing and traversing the contouring tree.

4. Architectural Modeling

The architectural modeling necessary to determine if a VLSI multiprocessor for real-time contour surface display generation is feasible is accomplished in two steps. The first step is the modeling of the algorithm component level (see Figure 11). The purpose of this step is to determine if the amount of code specified for the algorithm component computation is executable in real-time. In this step, an implementation of the algorithm component is analyzed. The analysis is performed in the context of a processor whose characteristics are similar to those of a general purpose microprocessor, the MC68000. The model constructed is a register transfer model of the algorithm component. In this model, the memory references that are made for each instruction's operation and for each operand's retrieval during the execution of the algorithm component are counted and recorded. Since the number of memory references a program makes is proportional to its run time, we only have to multiply by the amount of time a memory reference requires in order to obtain a measure of the real-time capability of the algorithm component processor ([Zyda,1981], [Zyda,1982], [Zyda,1983], [Zyda,1984a], [Aho,1974], and [Fuller,1977]). The value used in this study is 250 nsec per memory reference. This value is the slowest access time indicated for dynamic RAM (DRAM), and ROM chips announced over the last year in the IEEE journals *Computer*, and *Micro* (see Figure 12). Since there are access times indicated that are less than half that value, i.e. 70 nsec, we are conservative in the choice of 250 nsec as the time required to complete a memory reference.

The second step in the architectural feasibility modeling is the modeling of the total system of algorithm component processors (see Figure 13). The purpose of this step is to determine the total number of processors we can use in parallel, the load (number of algorithm components) per processor, and the total real-time capability of that system, i.e. the size of the largest three-dimensional grid for which we are able to generate the contour surface display in real-time. This part of the modeling effort extends the algorithm component modeling results to a model of the total system architecture for the real-time contour surface display generator. With the structure and real-time capability of the algorithm component processor established, we determine the capabilities of a system utilizing multiple copies of that processor. The parameters of the complete system modeled are derived from the requirements of the applications. The parameters utilized include such factors as the total size of the inputs and outputs, and the total number of algorithm components (and hence, the total number of algorithm component processors).

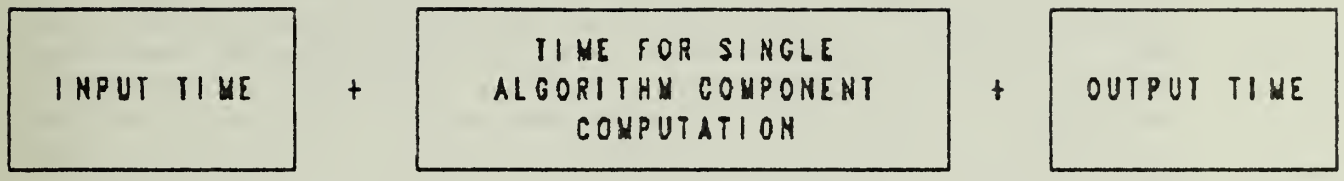


FIGURE 11
ALGORITHM COMPONENT ARCHITECTURAL MODEL

DRAM Chip Characteristics

Manufacturer	Chip	Chip Size	Access Time	Reference
AMD	AM9128	16K DRAM	70ns	Micro, Feb. 83
Mostek	MK45H64	64K DRAM	80ns	Micro, Aug. 83
				100ns
				120ns

ROM Chip Characteristics

Manufacturer	Chip	Chip Size	Access Time	Reference
Signetics	23256A	256K ROM	200ns	Computer, Jun. 83
Synertek	SY23128/A	128K ROM	200ns	Computer, Jul. 83
	SY23256/A	256K ROM	200ns	
American	S23128A	128K ROM	250ns	Computer, Jul. 83
Microsystems				

Figure 12
DRAM and ROM Chip Access Time taken from 1983 issues
of Computer, and IEEE Micro

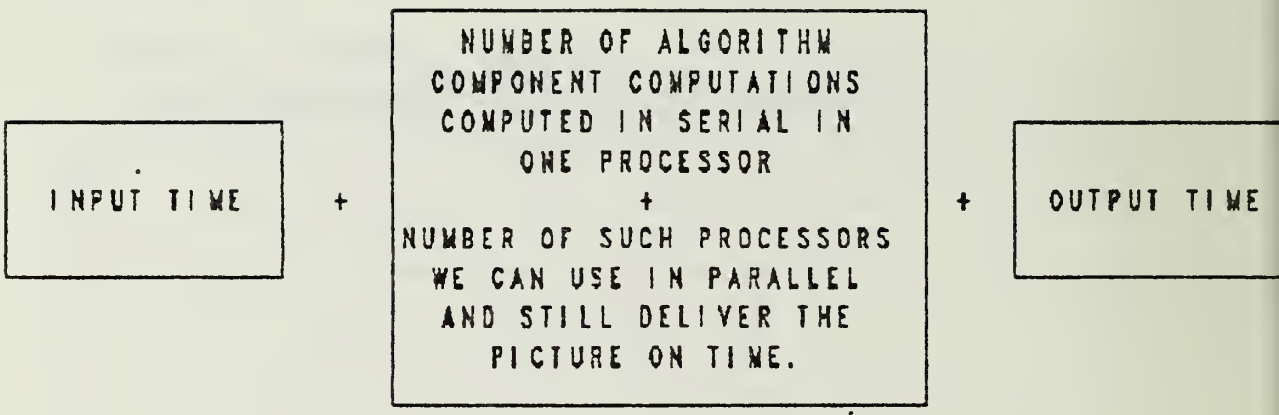


FIGURE 13
TOTAL SYSTEM ARCHITECTURAL MODEL

4.1. Architecture for the Algorithm Component

We begin the description of the architecture for the algorithm component with an overview diagram (see Figure 14). In that figure the important architectural pieces of the processor and their interconnections are depicted. The pieces shown are found in most processors. The important topics for our discussion are (1) the use of the hardware in the implementation of the algorithm component, and (2) the sizes of the hardware elements depicted in the figure.

In order to detail the sizes of the hardware elements in the figure, we first describe the operations expected of the algorithm component processor. There are only four: (1) reset the entire system of algorithm component processors, (2) accept a 2 x 2 subgrid description into a particular algorithm component processor, (3) place the coordinates generated for a particular 2 x 2 subgrid onto the system bus, and (4) generate the contours for the 2 x 2 subgrid held in the algorithm component processor. The first operation, the reset operation for the entire system of algorithm component processors, is clearly required. Computing systems are never constructed without some mechanism for providing a known initial state of the hardware.

The second operation, that of accepting a subgrid definition into a particular algorithm component processor, has implications for both the size of the RAM of the processor, and for the performance of its external communication mechanism. For that operation, the algorithm component processor needs to be able to recognize when a subgrid definition is addressed to it, and then needs to be able to store that information into its RAM. For both parts of this operation, we need to evaluate the size of the input to the algorithm component processor. This is accomplished by making a short list of the data input for a single algorithm component:

(1) 4 quantities for the grid values on the corners of the 2 x 2 subgrid (16 bytes)

(2) 2 values representing the lower lefthand coordinate of the 2 x 2 subgrid (2 bytes)

(3) 2 values representing the orthogonal coordinate and the orthogonal coordinate type (2 bytes)

(4) 1 value for the contour level (4 bytes)

If we assume 32-bit transfers to the algorithm component processor, this is a total of 6 references per 2 x 2 for the input operation, requiring an equivalent amount of RAM storage.

The third operation, that of placing the coordinates generated in a particular algorithm component processor onto the system bus, has implications similar to that of the input operation. For the output operation, the algorithm component processor needs to be able to recognize when it should deposit its coordinates onto the system bus, and needs to be able to provide RAM storage for those output coordinates beforehand. From [Zyda,1984a], we know that the largest output that can be generated for a 2 x 2 subgrid is 6 coordinate and drawing instruction quadruples (78 bytes). If we count the byte indicating the number of coordinates output, we need to perform 20 32-bit transfers for the output operation, and need to provide an equivalent amount of RAM storage.

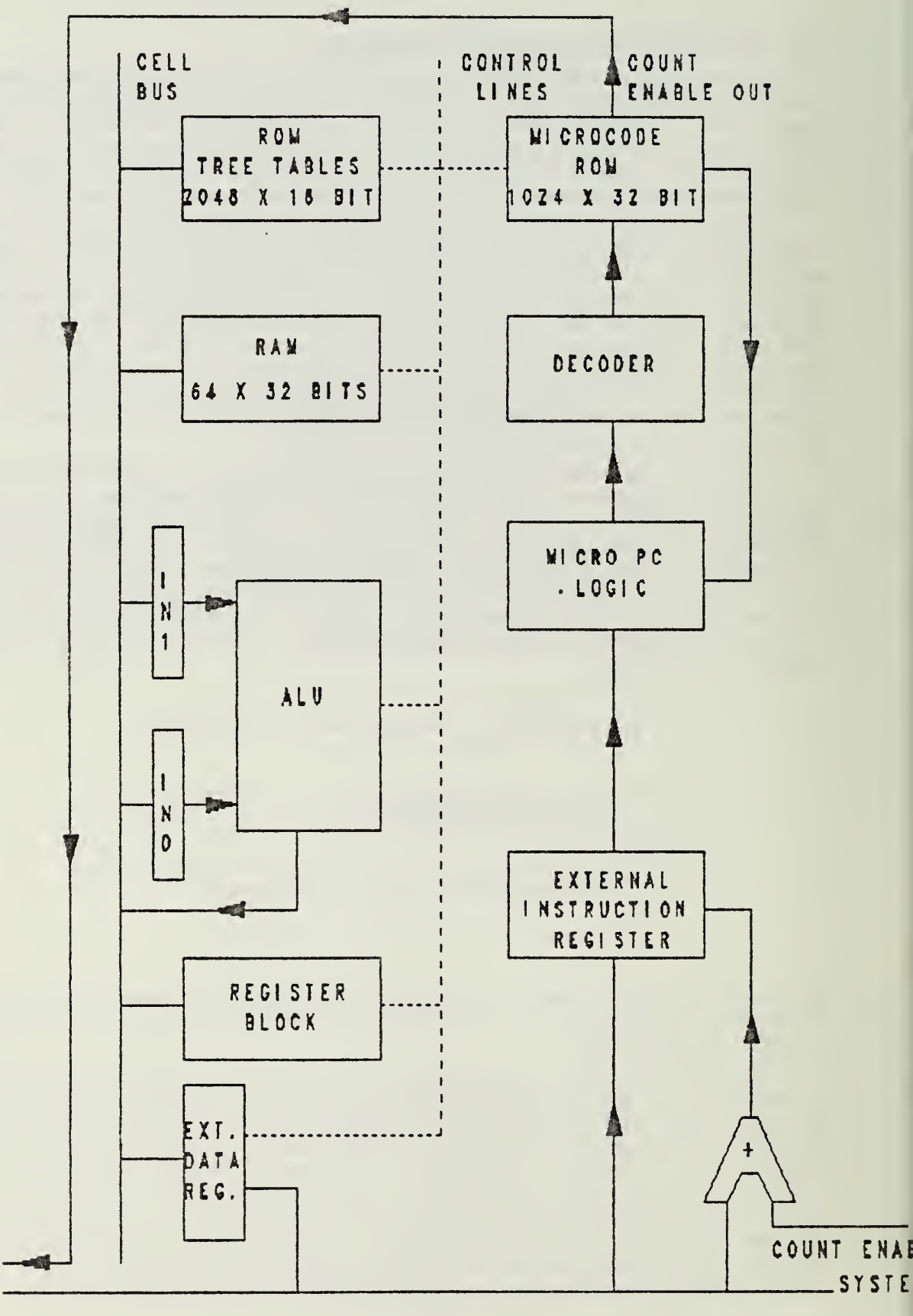


FIGURE 14
BLOCK DIAGRAM OF THE ALGORITHM COMPONENT PROCESSOR

The fourth operation, that of generating the contours for the 2 x 2 whose definition is held in the algorithm component processor, effects the size of all the memories in the algorithm component processor. If we use the algorithm simplifications described above, this means we need to provide space for the tree traversal list tables (2681 bytes), the algorithm component miscellaneous variables (45 bytes), and the code that performs the algorithm component computation (3080 bytes). (A comprehensive listing of all the data required in the algorithm component computation can be found in [Zyda,1984a], Figure 3.1.). The estimates for the input, output, tree traversal tables, and miscellaneous are derived directly from the data and data sizes required for the computation of the algorithm component. All the data sizes are rounded to the nearest byte, except for the large tree traversal tables where estimates are quoted in terms of the number of bits needed. The bit-wise specifications for the traversal tables are combined and then divided by the number needed to form a total specifiable in bytes.

The estimate for the size of the code required in the algorithm component processor is computed by totaling the number of instructions used in the four routines that comprise the register transfer model of that algorithm component. A value of four bytes per instruction is assumed. The values obtained for each of the four modeled routines are (1) 792 bytes for the control program of the contouring operation, (2) 500 bytes for computing the subgrid coordinates and average value point, (3) 304 bytes for computing the contouring tree configuration number, and (4) 1484 bytes for the traversal list usage and coordinate generation routine.

Combining the data and code totals, the algorithm component processor is seen to require 5909 bytes of storage, 148 bytes for input, output, miscellaneous, and temporaries (read/write memories), and 5761 bytes for the code and tree traversal lists (read-only memories). In our computation of the size of the algorithm component processor, the above values represent the space needed for registers, random-access and read-only memories. Space estimates for the rest of the hardware are not included. In order to provide a size value for the remainder of the architectural features in the algorithm component processor, we need to enumerate those hardware requirements.

The control portion of the algorithm component processor is shown in the right half of Figure 14. It is composed of the external instruction register, the microprogram logic, the decoder, and the microcode ROM. There is nothing special expected for this control section that is not standard among most processors. The only important feature is the relatively large microcode ROM that contains the actual contouring program. Above, we stated that this ROM required a minimum of 3080 bytes in order to be able to perform the expected operations. Rounding this to a power of two, and assuming horizontal microprogramming for the algorithm component processor, a 1024 by 32-bit memory is the estimate that is used in our VLSI feasibility determination.

Continuing with the topic of rounding the memory sizes and widths of the ROMs and RAMs specified on Figure 14, we find that the tree traversal ROM, originally specified as requiring a minimum of 2681 bytes, is best configured as 2048 by 16 bits. The reason for this large increase in the space requirement for the tree traversal tables is that the edge entries are expanded to 16 bits rather than the original 12 bits as specified above. The RAM of Figure 14, used to hold the subgrid definition, the coordinates generated, and any temporaries, is assumed to be 64 by 32-bits, up from the originally specified minimum of 148 bytes. We should note at this point that the ROMs and RAMs specified are expected to consume the majority of the area on the VLSI chip.

The ALU and the register block of Figure 14 are the remaining items for which we must develop a size estimate. The register block has no special requirements other than that there be about eight 32-bit registers. This is not a measured requirement but rather one suggested by the designs of other microprocessors. The ALU shown in Figure 14 is dealt with in the same way as the rest of the hardware in that we assume it too is little different than ALUs found in currently produced microprocessors. This means it has the capability to perform integer addition, integer subtraction, integer division, and integer multiplication. It would be nice to have floating point operations directly in the ALU but this is expensive and consumes considerable area on the VLSI chip. Any floating point operations we need to perform can be simulated using the integer arithmetic capabilities provided by this minimal ALU. It should be noted that the algorithm component of the contour surface display generation algorithm was originally implemented entirely with integer arithmetic.

4.2. Real-Time Capability of the Algorithm Component Processor

In order to determine if the amount of computation specified for the algorithm component is executable in real-time, one-thirtieth of a second, we need to put together a register transfer model of that algorithm and then to execute that model with the worst case inputs for the algorithm. As indicated above, a register transfer model counts the total number of memory references made by the algorithm component for both operation executions, and operand retrievals. There are four parts to the register transfer model of the algorithm component: (1) the input of the 2 x 2 subgrid to the algorithm component processor, (2) the output from the algorithm component processor, (3) the tree construction (traversal list indexing), and (4) the contour generation (traversal list usage). The memory reference count for each of these parts of the algorithm component needs to be modeled and totaled in order to determine the feasibility of executing in real-time a complete, worst-case set of input data.

The first part of the register transfer model is the number of memory references required to complete the input of the 2 x 2 subgrid to the algorithm component processor. The total number of 32-bit transfers for this operation was obtained in the previous section -- 6 32-bit transfers per 2 x 2 subgrid. The second part of the register transfer model, the number of memory references required to complete the maximum sized output, was also obtained in the previous section -- 20 32-bit transfers per 2 x 2 subgrid. The third part of the register transfer model, the tree construction (traversal list indexing), requires 602 32-bit references to (1) compute the center average value point from the four subgrid points (263 references), (2) determine if the points are in range of the current contour level (177 references), and (3) compute the configuration number (162 references). The fourth part of the register transfer model, the contour generation (traversal list usage), requires a maximum of 2048 32-bit references. It should be noted that this maximum is obtained for the subgrid that generates the maximum number of coordinate and drawing instruction quadruples, six quadruples per 2 x 2. For typical applications, the average number of coordinate and drawing instruction quadruples generated for the set of 2 x 2 subgrids that generate coordinates at all is 2.54 quadruples per 2 x 2. This value was obtained empirically through the monitoring of the execution of the contouring algorithm on several data sets typical of the expected applications. Though the use of this average number of coordinates could significantly lessen the number of memory references found for the contour generation (traversal list use) part of the register transfer model, the worst case of six quadruples, corresponding to 2048 memory references, must be used in the determination of the real-time capability of the algorithm component processor. The worst

case must be used because that case indicates the longest time the system of algorithm component processors will require for the completion of the contouring operation.

Once we have obtained the memory reference count for all four parts of the register transfer model of the algorithm component, we can total the memory references and determine if that component can be executed in real-time. For the register transfer model of the algorithm component, a total of 2676 memory references are required for (1) the input to the algorithm component processor (6 memory references), (2) the output from the algorithm component processor (20 memory references), (3) the tree construction, or traversal list indexing, for the 2 x 2 subgrid (602 memory references), and (4) the contour generation from the trees generated, or traversal lists indexed, (2048 memory references). At 250 nsec per reference, this is about 669 microseconds -- clearly under the one-thirtieth of a second (33,333 microseconds) goal we set for the algorithm component processor. In fact, given one-thirtieth of a second, we can accomplish about 50 algorithm component computations in serial. Now that we have established the feasibility of computing the algorithm component in real-time with the architecture proposed, we need to design a larger system of multiple algorithm component processors.

5. Larger System of Multiple Algorithm Component Processors

The first issue of importance that must be covered when considering the design of the larger system is the issue of how operations and data are communicated. Figure 15 contains a view of the proposed interconnection scheme for the algorithm component processors. In that figure, each processor is depicted as being connected to a system bus, and a serial control line called the count-enable line. As indicated in Figure 14, the system bus provides both data and instructions to the algorithm component processor. It also provides the pathway for data output back to the display controller. Not so clear in that figure is the function of the count-enable line. The count-enable line is a one bit control line that runs in a daisy-chain fashion from one algorithm component processor to the next. Its function is to provide a processor addressed capability for operations indicated to the larger system of processors. Its effect is to serialize the execution of processor addressed operations such as data input and output. This is accomplished in the following manner. Each algorithm component processor uses the logical OR of the global control line contained in the system bus and the count-enable line to determine if it should gate in the instruction currently presented on the system bus. A signal on the global control line indicates a global operation, and means that all processors of the system should perform the specified operation. Global operations are used to initiate the highly parallel computations of the algorithm component. A signal on the count-enable in line for an algorithm component processor indicates a processor addressed operation, and means that the instruction and any following data on the system bus are addressed to that specific processor. Once an algorithm component processor has gated in a processor addressed instruction and its data, it then sets the count-enable out line high. The setting of the count-enable out line to high indicates to the next processor in the chain that it should gate in the instruction and data next on the system bus. The count-enable mechanism is used to propagate processor addressed instructions throughout the system in an orderly fashion. Its effect is to serialize the execution of operations such as the input of data to and the output of data from each algorithm component processor.

It should be noted at this point that other processor interconnection schemes such as multiple buses for parallel data output have not been

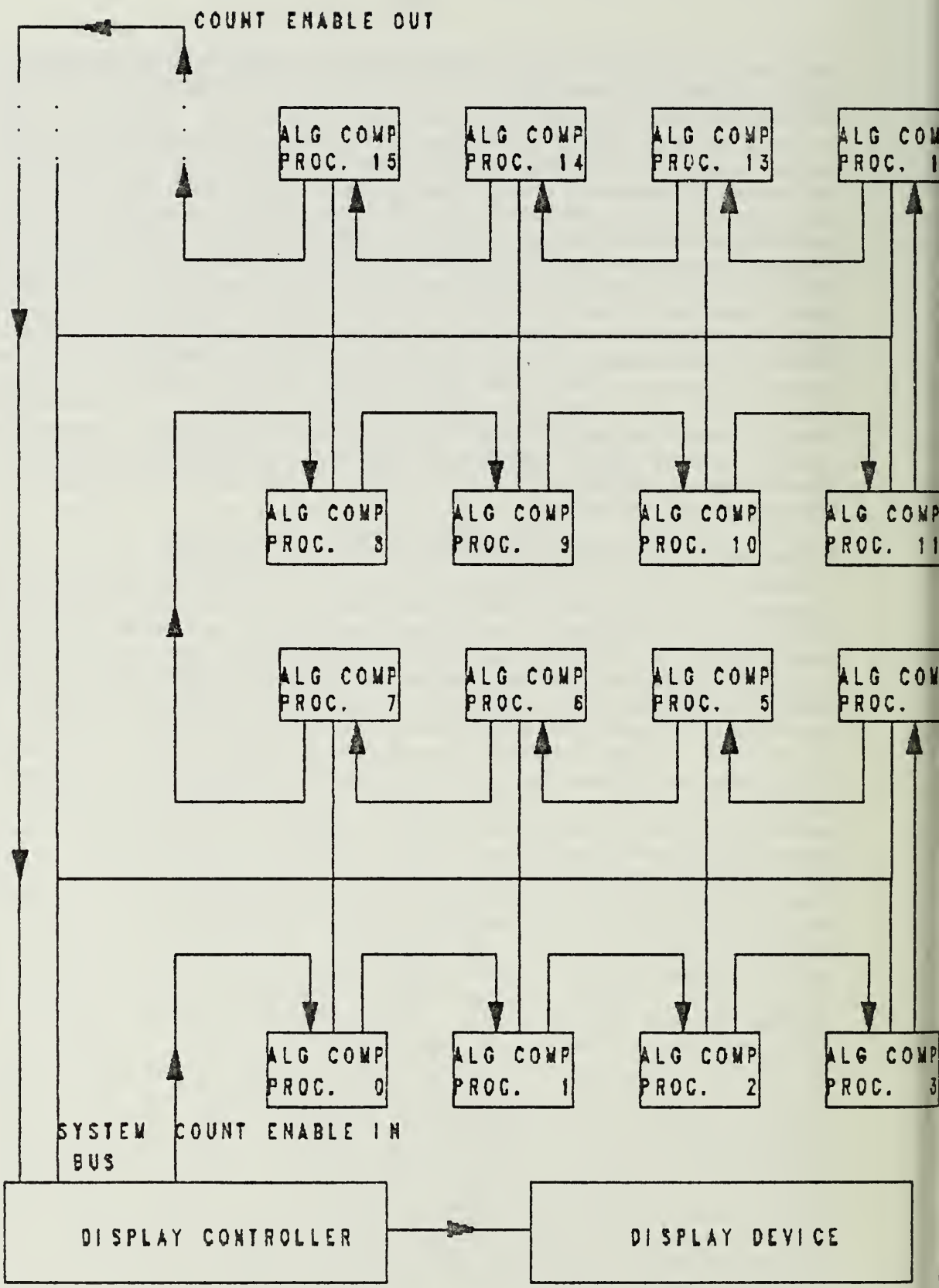


FIGURE 15
MULTIPLE ALGORITHM COMPONENT PROCESSOR INTERCONNECTIONS

considered in this study. The reason for this limitation is that the currently available display devices to which the output is directed, only have a single, 8 to 32 bit wide pathway for display list modification. The design of a display device with multiple, parallel pathways for display list modification is outside the scope of this study.

In order to complete our description of the communication mechanism for the system of multiple algorithm component processors, we need to estimate the widths of (1) the system bus data and control lines, (2) the count-enable lines, (3) the external instruction register, and (4) the external data register. The system bus and count-enable lines sizes are the most important because they extend across VLSI chip boundaries, and hence require package pins. The count-enable lines require two bits, one into and one out of each algorithm component processor. This requires two pins on the VLSI chip. The system bus specification is more difficult in that we have both data and control line widths to specify. The width of the data portion of the system bus is chosen to be 32 bits. This figure is based upon the number of pins we expect to be able to spare on the VLSI chip, and upon the fact that we assume a 32-bit processor, and 32-bit transfers in our register transfer models. In order to determine the width of the control line portion of the system bus, we need to compose a list of the signals we expect it to carry:

(1) global/processor addressed bit (1 bit)

(2) instruction bits (3 bits)

(3) data transfer control lines (6 bits)

(4) miscellaneous control lines (6 bits)

The sizes indicated for the data transfer and miscellaneous control lines are taken from the bus designs for similarly sized processors and are not exact [Hayes,1978]. The values quoted only serve as an estimate on the number of control signals expected. Consequently, the total estimate for the control portion of the system bus is 16 bits for a bus total of 48 bits. Adding the two pins for the count-enable lines, this means a minimum of 50 pins on the VLSI chip. This is somewhat under the current package limit of 64 pins, and allows room for additional pin requirements.

The sizes of the external data register and the external instruction register are set by the data width assignments made for the system bus. The instruction portion of the system bus was set at three bits based upon the fact that there are only four operations we expect to signal to the algorithm component processor. Consequently the external instruction register only needs three bits. The purpose of the external instruction register is to hold a signaled instruction until the control portion of the algorithm component processor is finished with its previous operation and ready to execute a new one.

The external data register is used to transfer data to/from the algorithm component processor from/to the data portion of the system bus. The data that is transfered into the algorithm component processor is data such as the subgrid definition and the new contour level. The data transfered out of the algorithm component processor is the set of coordinate and drawing instruction quadruples generated by the last execution of the generate contour instruction. Since the data width portion of the system bus is set at 32 bits, the external data register is also 32 bits. The initiation of data transfers through the external data register is carried out by the control section of the algorithm

component processor.

5.1. Modeling the Larger System of Algorithm Component Processors

The purpose of the model for the larger system of algorithm component processors is to answer the question of exactly how many algorithm component computations can be executed in parallel in one-thirtieth of a second, with the only limitation being that the coordinates and drawing instructions must be delivered within that same time period. For this model, we assume an infinite capability for processors. We also assume that to obtain the highest processor utilization, the individual processor may be responsible for multiple, serial algorithm component computations. The timing values for this step are obtained by extending the register transfer model developed for the algorithm component processor.

In order to determine the number of maximal algorithm component computations we can execute in parallel, we compose a model of that system:

Real-Time Available	=	Input Time	+	Computation Time	+	Output Time

The model forms a simple linear equation, with the real-time available on one side and the input, output, and computation times on the other. For this model, we make the following assumptions: (1) the amount of real-time available is 33.333×10^{-3} seconds, (2) all of the algorithm component computations occur in parallel, so only one maximal computation is added to the model's equation (2650 references @250 nsec/reference), (3) the only input is the single 32 bit new contour level, distributed to all processors via a global command (1 reference @250 nsec/reference), (4) the size of the output from each algorithm component computation is of average size (2.54 coordinates and drawing instruction quadruples, or 9 references, for each 2 x 2 subgrid that generates coordinates [Zyda,1984a]). The model has the following equation:

33.333 msec	=	1 ref @250 nsec per ref	+	2650 refs @250 nsec per ref	+	X(9 refs) @250 nsec per ref

The variable X stands for the maximum number of algorithm component computations that the modeled system can handle. Solving for X, we find that we can compute in parallel, in one-thirtieth of a second, 14,520 algorithm component computations, generating a total of 36,880 coordinate and drawing instruction quadruples. Again, this requires some 14,520 processors, each operating in parallel.

6. Further Applications Details

Once we have an idea of approximately how many algorithm component computations we can perform in one-thirtieth of a second, we then need to further examine the particular real-time application in order to determine if we are able to handle the expected maximum input data grid. Using the molecular modeling program presented above as the typical application, we find that the largest three-dimensional grid of interest is a cube of 30 units on each side [Barry,1979]. As discussed, a contour surface display is created for a three-

dimensional grid by generating the coordinates and drawing instructions for all possible orthogonal two-dimensional grids of that larger grid. For the 30 x 30 x 30 grid, this is 90 30 x 30 grids. Specifying this in total 2 x 2 subgrids, this is 75,690 2 x 2s that must be computed in one-thirtieth of a second. From our architectural discussion, we found that we have the capability for generating coordinates from 14,520 2 x 2 subgrids in one-thirtieth of a second. Given that this is considerably under the total number of 2 x 2s, there are several questions for which we must provide answers:

(1) For the applications of interest, what is the maximum number of 2 x 2s (of the 75,690 total) for which we expect to generate coordinates and drawing instructions?

(2) What is the maximum number of coordinates we expect to generate for those applications?

(3) How do we handle 2 x 2s that do not generate coordinates?

> (a) Do we send the 2 x 2 subgrids to the algorithm component processors each time a new contour level is set, eliminating non-productive 2 x 2s at a higher level?
>
> (b) or do we double up the processors we can handle with 2 x 2s of non-overlapping grid value ranges?

The first and second questions are related so we answer them by referring to studies of the applications of contour surface display generation. For those applications, we see that the maximum observed percentage of 2 x 2s that generate coordinates is 13 percent, or around 9900 2 x 2s. The number of coordinates generated for that system, the maximum number for our applications purposes, is 25,150 coordinate and drawing instruction quadruples. Clearly this is within the capabilities shown above for the system of algorithm component processors.

The third question, that of how we handle 2 x 2s that do not generate coordinates, is more difficult to answer. One possibility, as indicated in 3a, is to eliminate non-productive 2 x 2s at a higher level, sending only the coordinate productive ones to the algorithm component processors each time a new contour level is indicated. If we model this situation in a manner similar to that shown above, and assume an average number of coordinates generated for each 2 x 2, we find that the system can handle a maximum of 8712 2 x 2s in one-thirtieth of a second, not counting the time required for filtering out the non-productive 2 x 2s. This is not large enough to handle the maximal problem of 9900 2 x 2s computed in parallel though it is not a bad solution. The only problem with this solution is that it requires a higher level mechanism of some intelligence. We prefer to place all of the operations required for contour surface display generation into the multiprocessor system. If we were to build the multicomputer based upon this, it would require 8712 algorithm component processors of the type described above.

The second possibility, that mentioned in 3b, is to double up the algorithm component processors with 2 x 2s of non-overlapping grid point value range. Non-overlapping 2 x 2s never generate coordinates for the same contour level. If we keep track of the ranges for each 2 x 2, and the processor range in each algorithm component processor, we have a method for examining and computing coordinates for all 75,690 2 x 2s in roughly the same amount of time it takes

to perform the calculation on only those 2 x 2s that generate coordinates. The only question is can we find enough non-overlapping 2 x 2s in the typical problem to allow this solution? The answer is certainly we can. From studies of the value ranges of the grids we expect to encounter, we find that for a system of 75,690 2 x 2s the maximum number of non-overlapping partitions is about 16,000. This is an average of five 2 x 2 subgrids per partition, with an observed maximum of fifteen 2 x 2 subgrids in a single partition. Extrapolating these figures to the architecture, we find a requirement of 16,000 algorithm component processors, with a storage capacity of 15 2 x 2 subgrids in each processor.

The above has discussed one architecture for real-time contour surface display generation. The goal that guided the design of that architecture was the use of all of the parallelism available from the decomposition of the complete algorithm. There are clearly alternate architectures, not all of which can be discussed in this study. One such architecture is suggested by our original note, in the discussion of the real-time capability of the algorithm component processor, that each algorithm component processor could accomplish about 50 algorithm component computations in serial in one-thirtieth of a second. Before we can close the discussion of architecture for the contour surface display generator, we must consider a system of multiple algorithm component computations being performed in serial by a single algorithm component processor.

The model for such a system is easily composed from the data computed and derived for the highly parallel system. We will skip the preliminary considerations and model the system with the following assumptions. The input subgrids are already loaded into each algorithm component processor. The output from the total system of algorithm component processors is of average size, i.e. 2.54 coordinate and drawing instruction quadruples are generated from 9900 2 x 2 subgrids, for a total of 25,146 quadruples, or 89,100 memory references. The output is 32 bits wide, again due to the design of the display processor. In one-thirtieth of a second, there are 133,333 memory references using the figure of 250 nsec per memory reference. Subtracting the total number of memory references required for the output from the total number of memory references in one-thirtieth of a second, we find that 44,233 memory references are available for the computation of multiple subgrids in a single algorithm component processor. Dividing the total available computation time by the maximum amount of time an algorithm component processor could spend on a single algorithm component computation, 2650 memory references, we find that each algorithm component processor can compute the display for 16 subgrids in serial, with the system still being able to deliver the output in real-time. Dividing the total number of subgrids considered for our applications, 75,690 subgrids, by 16, we find that we need 4731 algorithm component processors.

Referring back to the discussion of our ability to coalesce the 75,690 subgrids into 16,000 partitions, each partition containing a maximum of 15 subgrids of non-overlapping grid values, we find that we really only have a requirement for 16,000 subgrid computations. If we design each algorithm component processor to hold 16 of these partitions, i.e. each processor has the capability for 15 times 16 subgrids, then we really only need 1000 processors. The only differences from the algorithm component processor previously described are (1) a larger RAM for the extra subgrid definitions, (2) a larger microcode ROM for the value range acceptance mechanism, and (3) a wider instruction portion of the system bus. The additional memory requirements are shown in Figure 16.

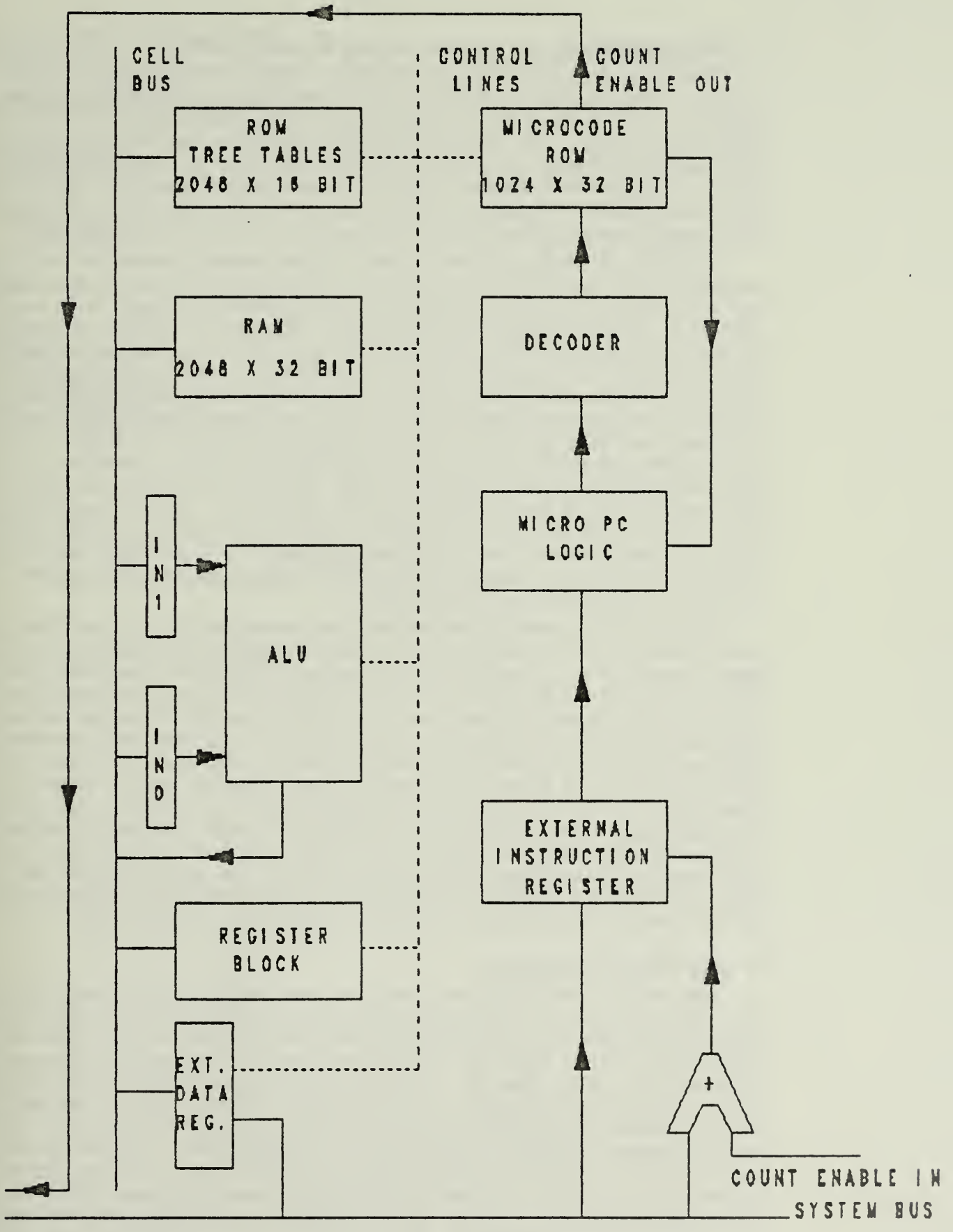


FIGURE 16
BLOCK DIAGRAM OF THE FINAL ALGORITHM COMPONENT PROCESSOR

7. VLSI Feasibility for the Contour Surface Display Generator

The above discussion has left us with an outline of the architecture necessary for real-time contour surface display generation. An important factor to consider at this point is the actual feasibility of implementing such a system in the VLSI technology. For this feasibility determination, we need to compute a value for the hardware complexity. The chief components of this complexity are the total number of transistors required, and the total number of VLSI chips. Once these values are obtained, we can then make a statement as to the feasibility of actually constructing the real-time contour surface display generator.

From the architectural specification, we can compute a value for the circuit complexity if we make some fairly simple assumptions. The first assumption is that if we obtain a circuit complexity for the algorithm component processor, then all we have to do to get the total system complexity is multiply by the total number of processors required. The second assumption is that the complexity of the algorithm component processor is less than or equal to the complexity of a known microprocessor, say perhaps the MC68000 used in our evaluation of the algorithm component's real-time capability. One paper, [Frank,1981], provides a comparison of the Motorola MC68000 and the Zilog Z8000 with figures for the total number of transistors. For the MC68000, the total transistor count is approximately 68,000, with 50,000 of those transistors being in the microcode ROMs and PLAs and the remaining 18,000 being in the registers and random logic. For the Z8000, the total transistor count is specified as 17,500. Consequently, a good estimate for the circuit complexity of a processor such as the one we propose for the algorithm component processor is 18,000 devices, not counting the RAM space, or the ROM space.

Figure 17 is a short table showing the breakdown of the algorithm component processor into pieces of similar circuit complexity. Using figures of two devices per bit for the random access memory (DRAM), and one device per bit for the read-only memory (ROM), we find that 195K devices are required for the storage alone. Adding that value onto the 18K devices that form the rest of the algorithm component processor, we note that the total number of devices the processor requires is on the order 215K. From the literature, we note that one million device VLSI chips are already being produced in the research lab [Gwynne,1983], with ten million device VLSI chips promised in the time period ranging from the year 1985 to the year 2001 [Uhr,1984]. This means 4 algorithm component processors per chip at the one million devices per chip level, and 48 algorithm component processors per chip at the ten million devices per chip level. For the 1000 processors needed for the contour surface display generator, this means a total system size in the range of 250 to 21 VLSI chips.

8. Large System Discussion

For most of the uniprocessor, von Neumann world a system design consisting of 1,000 processors seems infeasible. In fact, even systems of 50 interconnected processors are not viewed as particularly viable. A large part of this skepticism derives from the difficulties involved in early multicomputer attempts such as the Illiac IV, and the Carnegie-Mellon C.MMP and CM* projects ([Fuller,1977], [Barnes,1968], and [Wulf,1972]). These initial multicomputer efforts "peaked" at the level of around 50 processors. The focus of these projects has been to provide general purpose multicomputing. The economics of the design and construction effort dictated this slant. None of these multicomputers was particularly successful in fulfilling the need for general purpose multicomputing, and none of them was particularly useful for any specific application. Since the landmark 1977 article by Sutherland and Mead,

(1) RAM space -- (2 devices/bit)		
2048 x 32 bits	=	131,072 devices
(2) ROM space -- (1 device/bit)		
-- tree tables		
2048 x 16 bits	=	32,768 devices
-- microcode		
1024 x 32 bits	=	32,768 devices
(3) Rest of processor space --		
-- ALU		
-- register block		
-- control section		
-- external registers		
-- data and control buses		
	=	18,000 devices
Device total	=	214,608 devices (215K devices)

Figure 17
Algorithm Component Processor's Circuit Complexity Estimate

[Sutherland,1977], the economics and the focus of computer architecture research have changed. The VLSI revolution heralded by Sutherland and Mead has provided the capability for large scale, special architectures. One special architecture. the Massively Parallel Processor (MPP) delivered to NASA in December 1982, has 16,000 processors on 2,000 LSI chips [Potter,1983]. Its purpose is to solve large two-dimensional image processing applications in real-time. It is "general purpose" in the sense that it is good for a wide range of two-dimensional image processing applications, but it is still a special architecture. The contour surface display generator is an even more specialized architecture than the MPP, although it is just as feasible.

9. Conclusions

This study has focused on the architectural specification and feasibility determination of the real-time contour surface display generator. The conclusions we draw are that yes, we can put together such a multiprocessor. Once we have made such an assessment, we then need to consider the next steps in this research effort. Two directions come to mind, the second following directly from the first. The first direction concerns the details of how the real-time contour surface display generator is interfaced to a display system. The importance of this research direction becomes evident if we compute a value for the output data rate of the contour surface display generator. In Figure 15, the output is shown to be destined for a display device, with that output passing through a display controller. The assumption for that data transfer has been that it is accomplished via a DMA transfer mechanism of 32 bits width similar in operation to that of the DEC Unibus. Assuming that the output display is of average size, 89,100 32-bit memory references, this is a data rate of 10.7 megabytes per second. The delivery of data to the display system at the rate of 10.7 megabytes per second is somewhat faster than current display system technology allows. Compounded with this problem, is the fact that besides being able to deliver the picture within the given time constraints, we also need to maintain the functionality of the display system. This means that if we add the contour surface display generator to a display system that we cannot reduce or eliminate the display system's capability for real-time display rotation, scaling, translation, clipping, and other assorted, real-time operations. The full specification of the architectural changes required for the display system by the contour surface display generator are left as an area for further study.

Once we have answered the questions with respect to the contour surface display generator's impact on the design of the display system, the second research direction is to examine other graphics algorithms for implementation in VLSI. If we then perform the same study of the interfacing of those special purpose display generators with the display system, we can see if there are any general principles we can establish. It is not until this question is answered in the general case, that we can actually begin the systematic implementation in VLSI of special purpose, real-time display generators.

10. References

1. Aho, Alfred V., Hopcroft, John E., and Ullman, Jeffrey D. *The Design and Analysis of Computer Algorithms*. Reading, Massachusetts: Addison-Wesley Publishing Company, 1974, Chapters 1-5.

2. Barnes, G.H. et. al. "The ILLIAC IV Computer," *IEEE Transactions on Computers*, Vol. 17 (1968), pp. 746-757.

3. Barry, C.D. and Sucher, J. H. "Interactive Real-Time Contouring of Density Maps," American Crystallographic Association Winter Meeting, Honolulu, March 1979, Poster Session.

4. Faber, D.H., Rutten-Keulemans, E.W.M., and Altona, C. "Computer Plotting of Contour Maps: An Improved Method," *Computers & Chemistry*, Vol. 3, pp. 51-55, Great Britain: Pergamon Press Ltd., 1979.

5. Frank, Edward H. and Sproull, Robert F. "Testing and Debugging Custom Integrated Circuits," *Computing Surveys*, Vol. 13, No. 4 (December 1981), p. 425.

6. Fuller, S.H. et. al., "A Collection of Papers on CM*: A Multi-Microprocessor Computer System," Technical Report, Pittsburg: Carnegie-Mellon University, Department of Computer Science, February 1977.

7. Gwynne, P. "IBM's 512-Kbit Chip Sets Up Future Marketing Battle," *Industrial Research and Development*, Vol. 25, No. 11 (November 1983), p. 56.
8. Hayes, J.P. *Computer Architecture and Organization*, New York: McGraw-Hill Book Company, 1978, p. 408-418.

9. Newman, William H., and Sproull, Robert F. *Principles of Interactive Graphics*. Second Edition. New York: McGraw-Hill, 1979.

10. Potter, J.L. "Image Processing on the Massively Parallel Processor," *Computer*, Vol. 16, No. 1 (January 1983), p. 62.

11. Sutcliffe, D.C., "Contouring Over Rectangular and Skewed Rectangular Grids -- An Introduction," *Mathematical Methods in Computer Graphics and Design*, Edited by K.W. Brodlie, pp. 39-62, Great Britain: Academic Press, 1980.

12. Sutherland, I.E. and Mead, C.A. "Microelectronics and Computer Science," *Scientific American*, September 1977, pp. 210-228.

13. Uhr, Leonard. *Algorithm-Structured Computer Arrays and Networks*, Orlando, Florida: Academic Press, 1984.

14. Wright, Thomas and Humbrecht, John "ISOSRF -- An Algorithm for Plotting Iso-Valued Surfaces of a Function of Three Variables," *Computer Graphics: A Quarterly Report of SIGGRAPH-ACM*, Vol. 13, No. 2 (August 1979), pp. 182-189.

15. Wulf, W.A. and Bell, C.G. "C.MMP - A Multi-Mini-Processor," *AFIPS Conference Proceedings*, Vol. 41, Part II, FJCC, 1972, pp. 765-777.

16. Zyda, Michael J. *Algorithm Directed Architectures for Real-Time Surface Display Generation*, D.Sc. Dissertation, Dept. of Computer Science, Washington Univ, St. Louis, Missouri, January 1984a.

17. Zyda, Michael J. "A Decomposable Algorithm for Contour Surface Display Generation," Technical Report NPS52-84-xxx, Monterey, California: Department of Computer Science, Naval Postgraduate School, August 1984b.

18. Zyda, Michael J. "A Contour Display Generation Algorithm for VLSI Implementation," *Selected Reprints on VLSI Technologies and Computer Graphics*, Compiled by Henry Fuchs, p. 459, Silver Spring, Maryland: IEEE Computer Society Press, 1983.

19. Zyda, Michael J. "A Contour Display Generation Algorithm for VLSI Implementation," *Computer Graphics: A Quarterly Report of SIGGRAPH-ACM*, Vol. 16, No. 3 (July 1982), p. 135.

20. Zyda, Michael J. "Multiprocessor Considerations in the Design of a Real-Time Contour Display Generator," Technical Memorandum 42, St. Louis: Department of Computer Science, Washington University, December 1981.

21. Zyda, Michael J. "Joystick Driven Display Rotation and Control Console Management," Technical Memorandum 24, St. Louis: Department of Computer Science, Washington University, November 1980.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA	2
Office of Research Administration Code 012A Naval Postgraduate School Monterey, CA 93940	1
Chairman, Code 52Hq Department of Computer Science Naval Postgraduate School Monterey, CA 93940	20
Prof. Michael J. Zyda, Code 52Zy Department of Computer Science Naval Postgraduate School Monterey, CA 93940	40